ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2012) 297–305

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

Discrete first- and second-order adjoints and automatic differentiation for the sensitivity analysis of dynamic models☆

Ralf Hannemann[a,c,1], Wolfgang Marquardt[a], Uwe Naumann[b], Boris Gendler[b]

[a]*Aachener Verfahrenstechnik – Process Systems Engineering, RWTH Aachen University, 52056 Aachen, Germany*
[b]*LuFG Informatik 12: Software Tools for Computationial Engineering, RWTH Aachen University, 52056 Aachen, Germany*
[c]*German Research School for Simulation Sciences GmbH, 52425 Jülich, Germany*

Abstract

We describe the use of first- and second-order tangent-linear and adjoint models of the residual of linear-implicit autonomous differential algebraic systems in the context of an extrapolated Euler scheme. The derivative code compiler `dcc` is applied to a C-implementation of the residual to get first derivative code. Second-(and higher-)order derivative models are obtained by reapplication of `dcc` to its own output. The resulting solver serves as a first proof of concept of a new platform for source-level manipulation of mathematical models that is currently under development at RWTH Aachen University.

Keywords: first-order adjoints, second-order adjoints, discrete adjoints, one step extrapolation, parametric differential-algebraic equations

1. Introduction

The general interest in mathematical modeling and numerical simulation has been growing with the ability to handle increasingly complex scientific problems over the past decades. Nowadays, simulations are able to replace real life experiments in many situations where these are either not possible (e.g. operator training for extreme situations in chemical or power plant operation) or too expensive (simulation of crash behavior of cars). Thus, the ability to obtain reliable information based on simulations is a key factor for success in industry as well as academia. Very high-level modeling languages such as Modelica[2], VHDL[3], or gPROMS[4] have been developed to support the user in formulating the mathematical model. These tools allow to formulate models in an intuitive, equation-based representation. The facilities of these descriptive modeling languages also allow for hierarchical, highly structured object-orientated modeling. All the aforementioned issues are related to model development and maintenance. While model simulation has

☆This work was supported by the German Research Foundation under grant MA 1188/28-1 and by the ERS program from RWTH Aachen University.

Email addresses: `ralf.hannemann@avt.rwth-aachen.de` (Ralf Hannemann), `wolfgang.marquardt@avt.rwth-aachen.de` (Wolfgang Marquardt), `bgendler@stce.rwth-aachen.de` (Uwe Naumann), `naumann@stce.rwth-aachen.de` (Boris Gendler)

[1]Corresponding author

[2]`http://www.modelica.org/`

[3]`http://www.eda.org/vhdl-200x/`

[4]`http://www.psenterprise.com/gproms/`

1877-0509
doi:10.1016/j.procs.2010.04.033

been the primary focus for the interpretation of the model equations, their use in optimization problems for different kinds of applications is becoming increasingly important in scientific as well as industrial applications. These optimization problem formulations include for instance (i) model parameter estimation, (ii) optimal design of experiments for parameter estimation or model structure discrimination, (iii) design optimization, (iv) optimization of operations, (v) soft sensing and process monitoring, and (vi) model-predictive control and real-time optimization.

The solution of each of these simulation and optimization problems typically requires the evaluation of symbolic expressions involving different type and order of derivatives. Most of the solution engines linked to a modeling tool focus on one or few specific tasks - often just simulation - and do not provide derivative information required by other tasks. Classical numerical approximation using finite difference approximations is often computationally expensive and may be even infeasible for a large number of variables. Moreover, this approach yields a number of disturbing numerical effects making the quality of the obtained approximation highly questionable. A semantic code transformation technique known as *Automatic Differentiation* (AD) [1] allows for derivatives of arbitrary order to be computed efficiently and with machine accuracy. AD has been applied successfully to a large number of problems in science and engineering [2, 3, 4]. *Adjoint* versions of the original simulation F (with $F : \mathbb{R}^{n_{\text{in}}} \to \mathbb{R}^{n_{\text{out}}}$ being the operator mapping the data specifying the simulation experiment into the desired simulation results) are used to compute gradients ($n_{\text{out}} = 1$) of size n_{in} at a small constant multiple of the computational cost $Cost(F)$ of the simulation itself.[5]

The aim of the AC-SAMMM project[6] is to provide a tool that allows the calculation of specified symbolic expressions involving any kind of derivative based on AD using code transformation for a structured model represented in any equation-oriented modeling language. This paper describes a first successful integration of a software tool for the solution of parametric differential-algebraic systems into the AC-SAMMM infrastructure.

2. Parametric differential-algebraic systems

Parametric differential-algebraic systems arise in many engineering applications. They describe kinetics of chemical reactions [5], complete chemical manufacturing processes [6], mechanical multibody systems [7], electrical circuits [8] or even biological systems [9]. In general, these systems cannot be solved analytically such that numerical solution methods are required. The mathematical problem definition is as follows: Let $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ and $x_0 : \mathbb{R}^m \to \mathbb{R}^n$ be sufficiently smooth, $M \in \mathbb{R}^{n \times n}$ be a possibly singular matrix. Let $p \in R^m$ be a constant parameter vector and $I = [0, T]$ be a time interval. We consider the linear-implicit autonomous differential-algebraic equation

$$M\,\dot{x} = f(x, p), \tag{1}$$

where we assume the index of eq. (1) to be less or equal to one. Further, we assume that x_0 provides consistent initial conditions for eq. (1). The aim is to find a continuously differentiable function $x : I \times \mathbb{R}^m \to \mathbb{R}^n$ that solves the initial value problem (IVP)

$$x(0, p) = x_0(p), \tag{2}$$

$$M\,\dot{x}(t, p) = f(x(t, p), p) \quad \forall\, t \in I. \tag{3}$$

Under weak assumptions on f and x_0 the solution $x(t, p)$ is multiple continuously differentiable with respect to p, which we assume in the following. For details on the existence theorems and mathematical properties of differential-algebraic systems we refer to [8]. For a couple of computational tasks like parameter estimation [5] or dynamic optimization [10] not only the solution of eq. (1) is required, but also the evaluation of a so-called objective function $\phi : \mathbb{R}^n \to \mathbb{R}^k$ to be evaluated at $x(T, p)$ and thus resulting in an objective function $\Phi : \mathbb{R}^m \to \mathbb{R}^k$ which is only dependent from $p \in \mathbb{R}^m$:

$$\Phi(p) = \phi(x(T, p)) \in \mathbb{R}^k. \tag{4}$$

[5] ... as opposed to a computational complexity of $O(n_{\text{in}}) \cdot Cost(F)$ for finite differences or the *tangent-linear* model.

[6] See also `http://wiki.stce.rwth-aachen.de/bin/view/Projects/ERS/WebHome`.

E. g. Φ could be sum-of-squares objective of a parameter estimation problem that shall be minimized. If the minimization is carried out by second-order gradient-based method the optimizer also requires the first and second derivatives of Φ with respect to p, namely

$$\Phi_p(p) = D_p\,\phi(x(T,p)), \quad \text{and} \quad \Phi_{pp}(p) = D_{pp}\,\phi(x(T,p)), \tag{5}$$

where we use subscripts to denote the partial derivative of a function and the D_p notation for the total derivative operator with respect to p. If Φ represents such an objective function, we have $k = 1$ which, for notational convenience, we assume in the following. The generalization to $k > 1$ is straightforward. Here, the operator $F : \mathbb{R}^{n_{\text{in}}} \to \mathbb{R}^{n_{\text{out}}}$ corresponds to ϕ, where $n_{\text{in}} = m$ and $n_{\text{out}} = k$.

3. NIXE – a solver for differential-algebraic systems

NIXE is a C++ template class for the solution of eqs. (2) and (3) as well as for the efficient computation of the gradients and Hessians in eq. (5). The solver relies on the extrapolation of the linearly-implicit Euler discretization [11, 12] which gives reason to its name: NIXE, an acronym for **N**IXE **I**s e**X**trapolated **E**uler. The concepts of the algorithm are briefly sketched in the following.

The algorithm creates a series of basic step sizes $H_1, \ldots, H_N$ and approximations for the states $x^0, x^1, \ldots, x^N$ at time instances $t_0, t_1, \ldots, t_N$, such that

$$t_i = \sum_{l=1}^{i} H_i, \quad t_N = T, \quad x^i \approx x(t_i, p), \quad i = 0, 1, \ldots, N, \tag{6}$$

where by abuse of notation the right superscripts of x are indices, not powers. We sketch the step from x^{i-1} to x^i. Let $j_{\max}(i)$ be the maximal extrapolation order in this step. For $j = 1, 2, \ldots, j_{\max}(i)$ and an increasing sequence of integers $n_1 < n_2 < n_{j_{\max}(i)}$ we set $h_j := H_i/n_j$, $x_{j,0} := x^{i-1}$, $p_{j,k} := p$ and apply the scheme

$$x_{j,k+1} = x_{j,k} + (M - h\,A_0)^{-1} h_j\, f(x_{j,k}, p_{j,k}), \quad j = 1, \ldots j_{\max}(i),\ k = 0, \ldots n_j - 1, \tag{7}$$

where

$$A_0 = \frac{\partial f}{\partial x}(x^{i-1}, p). \tag{8}$$

This discretization scheme is of order 1 and provides us with a family of first-order approximations x_{j,n_j} for x^i. One applies extrapolation to achieve higher orders [11]. The basic step sizes H_i and the extrapolation order $j_{\max}(i)$ are determined on-line by a combined step size and order control. The reader is referred to [11] or [12, pp. 131–141] for a more detailed presentation.

Since the linearly-implicit Euler discretization is a so-called W-method [12, p. 114] the Jacobian A_0 could also be replaced by an approximation [13]. This fact is exploited in the numerical computation of the first-order sensitivities

$$s_i(t,p) := \frac{\partial x}{\partial p}(t,p),\ i = 1, \ldots, m,$$

which satisfy the initial value problem

$$M\,\dot{s}_i(t,p) = \frac{\partial f}{\partial x}(x(t,p),p)\,s_i(t,p) + \frac{\partial f}{\partial p_i}(x(t,p),p), \tag{9}$$

$$s_i(0,p) = \frac{\partial x_0}{\partial p_i}(p). \tag{10}$$

We set

$$\mathbf{M} := \operatorname{diag}(M, \ldots, M),\ \mathbf{y} := \begin{pmatrix} x \\ s_1 \\ \vdots \\ s_m \end{pmatrix},\ F(\mathbf{y}, p) := \begin{pmatrix} f(x,p) \\ f_x(x,p)\,s_1 + f_{p_1}(x,p) \\ \vdots \\ f_x(x,p)\,s_m + f_{p_m}(x,p) \end{pmatrix},\ \mathbf{y}_0(p) := \begin{pmatrix} x_0(p) \\ x_{0\,p_1}(p) \\ \vdots \\ x_{0\,p_m}(p) \end{pmatrix}.$$

The initial value problems in eqs. (2)-(3) and eqs. (9)-(10) can be solved as one system:

$$\mathbf{M}\,\dot{\mathbf{y}}(t,p) = F(\mathbf{y}(t,p),p), \tag{11}$$

$$\mathbf{y}(0,p) = \mathbf{y}_0(p). \tag{12}$$

If the linear-implicit Euler discretization is used to solve eqs. (11) and (12) it turns out that the resulting numerical computations are identical if the tangent-linear mode of automatic differentiation is applied to eq. (7). Hence, in this case we have the equivalence

first differentiate, then discretize ⇔ first discretize, then differentiate.

The right hand side of eq. (11) includes already first-order derivatives of $f(x(t,p),p)$. Hence, The Jacobian of $F(\mathbf{y}(t,p),p)$ involves second derivatives. However, Schlegel et al. [14] exploit the already mentioned W-method property to show that these second-order derivative can be ignored while still preserving the convergence to the sensitivities. NIXE also implements this reduced method.

If k in eq. (4) is small, e.g. $k = 1$ in our assumed case of an objective function in parameter estimation, it may be more efficient to apply adjoint sensitivity analysis. NIXE implements a modified discrete-adjoint approach. It applies the reverse mode of automatic differentiation to eq. (7) but ignores appearing second-order derivatives. Denoting $\lambda_{j,k} \approx \bar{x}_{j,k}$ and $\nu_{j,k} \approx \bar{p}_{j,k}$ the modified discrete adjoint vectors we get

$$\lambda_{j,k} = \lambda_{j,k+1} + h_j\,\lambda_{j,k+1}^T (M - h\,A_0)^{-1} f_x(x_{j,k},p), \quad j = 1,\dots j_{\max}(i),\ k = n_j - 1,\dots,0, \tag{13}$$

$$\nu_{j,k} = \nu_{j,k+1} + h_j\,\nu_{j,k+1}^T (M - h\,A_0)^{-1} f_p(x_{j,k},p), \quad j = 1,\dots j_{\max}(i),\ k = n_j - 1,\dots,0. \tag{14}$$

The discrete adjoints evolve backwards in time and indices. If x^N is the numerically computed value for $x(T,p)$, the final values of the modified discrete adjoints are given by $\lambda^N = \phi_x(x^N)$ and $\nu^N = \phi_p(x^N) = 0$. This modified discrete approach yields derivatives which are, up to rounding errors, identical to the derivatives computed by Schlegel et al. [14]. As in the case of first-order adjoints, second derivatives are computed by a modified discrete second-order adjoint scheme with similar efficiency-increasing reductions. The second summands in eqs. (13) and (14) involve vector-Jacobian products. This motivates us to introduce the Hamiltonian

$$H : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}, \quad H(x,\lambda,p) := \lambda^T f(x,p), \tag{15}$$

such that the vector-Jacobian products in eqs. (13) and (14) can be realized by evaluating first-order partial derivatives of H with respect to x and p.

The overall computation of Φ, Φ_p and Φ_{pp} consists of two steps.

1. One forward sweep which solves for $x(T,p)$, the sensitivities $s_i(T,p)$, $i = 1,\dots,m$, and the objective function value $\Phi(p)$. At a couple of checkpoints the states and sensitivities are store for the subsequent backward sweep.
2. One backward sweep which implements the modified discrete second-order adjoint sensitivity analysis to compute $\Phi_p(p)$ and $\Phi_{pp}(p)$.

Aside from computational routines which are called only once to evaluate the initial and final values for the states, sensitivities, adjoints and second-order adjoints, a couple of derivative and derivative projections of $f(x,p)$ and $H(x,\lambda,p)$ are evaluated several times during the numerical computations. During the forward sweep, these are the following derivatives:

$$f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n, \quad f_x : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^{n\times n}, \quad f_p : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^{n\times m}. \tag{16}$$

During be backward sweep, the derivatives in (16) are required to locally reconstruct the forward solution. Additionally, the following derivatives are evaluated:

$$H_x : \quad \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n, \quad H_p : \quad \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m, \tag{17}$$

$$D_p H_x : \quad \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^{n\times m} \times \mathbb{R}^{n\times m} \to \mathbb{R}^{n\times m}. \tag{18}$$

$$D_p H_p : \quad \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^{n\times m} \times \mathbb{R}^{n\times m} \to \mathbb{R}^{m\times m}, \tag{19}$$

Table 1: Forward sweep

Set $x^0 := x_0(p)$ and $s^0 := (D_p\, x_0)(p)$.
for $i = 0, \ldots, N-1$ **do**
  Compute the Jacobian $A_0 = f_x(x^i, p)$
  for $j = 0, \ldots, j_{\max}(i)$ **do**
    Set $h_j := H_i/n_j$
    Compute the LU-decomposition $LU = M - h_j A_0$
    Set $x_{j,0} := x^i$
    for $k = 0, \ldots, j-1$ **do**
      Compute $f(x_{j,k}, p)$, $f_x(x_{j,k}, p)$ and $f_p(x_{j,k}, p)$, i.e. derivatives of eq. (16)
      Set $x_{j,k+1} := x_{j,k} + h_j\,(LU)^{-1} f(x_{j,k}, p)$
      Set $s_{j,k+1} := s_{j,k} + h_j\,(LU)^{-1}\left(f_x(x_{j,k}, p)s_{j,k} + f_p(x_{j,k}, p)\right)$
    end for
  end for
  Compute x^{i+1} and s^{i+1} by means of extrapolation
end for
Set $\Phi(p) := \phi(x^N)$

where $\lambda = \lambda(p)$ is the adjoint vector. Employing NIXE, the derivatives marked by the preceding total derivative operator D_p have to be provided as projections by the user. All other derivatives have to be provided as whole matrices, where the interfaces for f_x and f_p support dense and sparse storage schemes.

The detailed schedule of the forward sweep is given in Table 1. Note, that this presentation of the algorithm exhibits redundant function evaluations which are avoided in practice.

The backward sweep is sketched in detail in Table 2. Here we denote with $\mu = D_p\,\lambda$ and $\xi = D_p\,\nu$ the matrices of the second-order adjoints. Note, that the backward sweep requires information of the forward sweep which is restored from a couple of checkpoints.

4. The Derivative Code Compiler `dcc`

The derivative code compiler `dcc` is a tool for automatic differentiation of C- code[7] by semantic source code transformation. For a given implementation of $f(x, p)$ as $f = g(x, p)$ in C- `dcc` generates code for the computation of derivatives of arbitrary order. Higher derivatives are generated by reapplication of `dcc` to its own output. In particular, `dcc` generates tangent-linear, first- and second-order adjoint models.

The *tangent-linear model* $f^{(1)} = g^{(1)}\left(x, x^{(1)}, p, p^{(1)}\right) \in \mathbb{R}^n$ where

$$g^{(1)} \equiv f'(x, p) \cdot \begin{pmatrix} x^{(1)} \\ p^{(1)} \end{pmatrix}, \tag{20}$$

is used to compute the Jacobian-vector product $f_x(x, p)\, x^{(1)} + f_x(x, p)\, p^{(1)}$. Note that we use parenthesized upper indices $^{(i)}$ to denote tangent-linear projections and parenthesized lower indices $_{(i)}$ to mark adjoint projections. The number i indicates that the variable is automatically created by the ith application of `dcc`.

The *adjoint model* $\begin{pmatrix} x_{(1)} \\ p_{(1)} \end{pmatrix} = g_{(1)}\,(x, x_{(1)}, p, p_{(1)}, f_{(1)}) \in \mathbb{R}^{n+m}$ where

$$g_{(1)} \equiv \begin{pmatrix} x_{(1)} \\ p_{(1)} \end{pmatrix} + (f'(x, p))^T \cdot f_{(1)}, \tag{21}$$

[7] Our focus is on a subset of C/C++, from hereon referred to as C-, that is rich enough to cover the fundamental elements of numerical simulation codes.

Table 2: Reverse sweep

Set $\lambda^N := \phi_x(x^N)$, $\nu^N := 0$, $\mu^N := (D_p\,\phi_x)(x^N, s^N, p)$and $\xi^N := 0$
for $i = N-1, \ldots, 0$ **do**
  Compute the Jacobian $A_0 = f_x(x^i, p)$
  for $j = 0, \ldots, j_{\max}(i)$ **do**
    Set $h_j := H_i/n_j$
    Compute the LU-decomposition $LU = M - h_j A_0$
    Set $\lambda_{j,n_j} := \lambda^i$, $\nu_{j,n_j} := \nu^i$ $\mu_{j,n_j} := \mu^i$ and $\xi_{j,n_j} := \xi^i$
    for $k = j-1, \ldots, 0$ **do**
      Compute $\hat{\lambda} = (LU)^{-1}\lambda_{j,k+1}$ and $\hat{\mu} = (LU)^{-1}\mu_{j,k+1}$
      Compute the derivatives of eqs. (17) – (19) evaluated at $(x_{j,k}, \hat{\lambda}, p, s_{j,k}, \hat{\mu})$
      Do some work to compute $\lambda_{j,k}$, $\nu_{j,k}$, $\mu_{j,k}$ and $\xi_{j,k}$
    end for
  end for
  Compute λ^i, ν^i, μ^i and ξ^i by means of extrapolation
end for
Set $\Phi_p(p) := (\lambda^0)^T (x_0)_p(p) + \nu^0$ and $\Phi_{pp}(p) := (\lambda^0)^T (x_0)_{pp}(p) + (\mu^0)^T (x_0)_p(p) + \xi^0$

computes $(\hat{x}^T, \hat{p}^T) + \lambda_1^T \left(f_x(x,p), f_p(x,p)\right)$, which is stored in the variables $x_{(1)}$ and $p_{(1)}$ at the exit of the routine. Here, $\lambda_1 = f_{(1)}$ is the seeding vector and $\hat{x}^T$ and $\hat{p}$ take the initial values of $x_{(1)}$ and $p_{(1)}$ at the entry of the routine.

The *second-order adjoint model*

$$\begin{pmatrix} x^{(2)}_{(1)} \\ p^{(2)}_{(1)} \end{pmatrix} = g^{(2)}_{(1)}\left(x, x^{(2)}, x^{(2)}_{(1)}, p, p^{(2)}, p^{(2)}_{(1)}, f_{(1)}, f^{(2)}_{(1)}\right) \in \mathbb{R}^{n+m}$$

is defined by

$$g^{(2)}_{(1)} \equiv \begin{pmatrix} x^{(2)}_{(1)} \\ p^{(2)}_{(1)} \end{pmatrix} + (f'(x,p))^T \cdot f^{(2)}_{(1)} + < f_{(1)}, f''(x,p), \begin{pmatrix} x^{(2)} \\ p^{(2)} \end{pmatrix} >, \tag{22}$$

where the second term is the projection of the $(n \times (n+m) \times (n+m))$-Hessian tensor $f''(x,p)$ in directions $f_{(1)} \in \mathbb{R}^n$ and $\begin{pmatrix} x^{(2)} \\ p^{(2)} \end{pmatrix} \in \mathbb{R}^{n+m}$. Note that these projections are uniquely defined due to the symmetry of $f''(x,p)$ in the $(n+m)$-dimensions. Please also note that (20)-(22) have to be interpreted as assignments, especially $(x_{(1)}, p_{(1)})^T$ in (21) and $(x_{(2)}, p_{(2)})^T$ in (22) serve both as input and output variables, which are usually initialized to zero by the user.

The tangent-linear model is used to compute both $f_x(x,p) \in \mathbb{R}^{n \times n}$ and $f_p(x,p) \in \mathbb{R}^{n \times m}$. Let $f(x,p)$ be implemented by a C-routine with the following signature

```
void g(int n, int m, double* x, double* p, double* f)
```

where the inputs x and p are n- and m-vectors, respectively, and where the n outputs are returned in the vector f. `dcc` generates a tangent-linear version of g with the following signature:

```
void d1_g(int n, int m, double* x, double* d1_x,
    double* p, double* d1_p, double* f, double* d1_f)
```

The superscript in $v^{(1)}$ is represented by a corresponding prefix d1_v. The actual source for d1_g must be omitted due to space restriction. Refer to the AC-SAMMM website[8] for examples of first- and higher-order derivative codes generated automatically by `dcc`.

$f_x(x,p)$ is accumulated by letting d1_x range over the Cartesian basis vectors in $\mathbb{R}^n$. The vector d1_p is set to zero. d1_f contains the product of the Jacobian of f with respect to both x and p with $\dot{x}$ =d1_x and $\dot{p} = 0$. Sparsity in $f_x(x,p)$

[8] www.stce.rwth-aachen.de

should be exploited, for example, through direct Jacobian compression techniques [15]. Total derivatives in directions corresponding to sums of Cartesian basis vectors representing *structurally orthogonal* columns[9] in f_x are computed in this case. Similarly, $f_p(x, p)$ is accumulated by letting d1_p range over (sums of) the Cartesian basis vectors in $\mathbb{R}^m$. The vector d1_x must be set to zero. Consequently, d1_f contains the product of the Jacobian of f with respect to both x and p with $\dot{x} = 0$ and $\dot{p}$ =d1_p.

A total of $n + m$ runs of d1_g is required to compute f_x and f_p in tangent-linear mode. Depending on the actual values for n and m it might be computationally less expensive to perform the at most n runs of the adjoint model to accumulate f_x and f_p as products of the transposed Jacobian of f with respect to both x and p with (sums of) Cartesian basis vectors in $\mathbb{R}^n$. Typically the adjoint code generated by `dcc` yields a constant overhead of at least two when compared to the tangent-linear code. The product of the transposed Jacobian with an n-vector takes roughly twice as long as the product of the Jacobian with an $n - m$-vector due to the data flow reversal mechanism that is inherent to any adjoint code. Refer to [16] for details. For most of our current target applications we observe that $n \approx m$. Hence, the computational costs of accumulating f_x and f_p in tangent-linear or adjoint mode are roughly identical.

The adjoint model of f is used to compute both $H_x(x, p, \lambda) = \lambda^T f_x(x, p) \in \mathbb{R}^n$ and $H_p(x, p, \lambda) = \lambda^T f_p(x, p) \in \mathbb{R}^m$. `dcc` generates an adjoint version of g with the following signature:

```
void b1_g(int n, int m, double* x, double* b1_x,
  double* p, double* b1_p, double* f, double* b1_f)
```

The subscript in $v_{(1)}$ is represented by a corresponding prefix b1_v. For use with NIXE we set b1_f=λ and b1_x=b1_p=0.

`dcc` supports interprocedural data flow reversal based on subroutine argument checkpointing also known as *joint call tree reversal* [1]. Arbitrary checkpointing schemes can be implemented by the user through appropriate restructuring of the original source code. Code for the storage and recovery of the subroutine arguments needs to be provided by the user as its automatic generation at compile time constitutes an in general undecidable problem due to missing array descriptors in C/C++. Fortran allows for a higher level of automation. Moreover, the user has to provide upper bounds on the sizes of the global data and control flow reversal stacks that are allocated statically by `dcc`. Other derivative code compilers, such as, for example, OpenAD [17] or Tapenade [18] implement dynamically growing stacks. While this approach is more convenient for the user it complicates the generation of higher-order adjoint codes by reapplication of the derivative code compiler to its own output. Algorithmically the two approaches are similar. A more detailed discussion of the generated adjoint code is beyond the scope of this paper.

The second-order adjoint model of f is used to implement both $D_pH_x = H_{xx}x^{(2)} + H_{x\lambda}\lambda^{(2)} + H_{xp}$ and $D_pH_p = H_{px}x^{(2)} + H_{p\lambda}\lambda^{(2)} + H_{pp}$, where $x^{(2)}$ and $\lambda^{(2)}$ can be interpreted as the partial derivatives $(\partial/\partial p)x(t, p)$ respectively $(\partial/\partial p)\lambda(t, p)$ evaluated at the current time t[10]. Reapplication of `dcc` in tangent-linear mode to the previously generated adjoint code yields a second-order adjoint version of g with the following signature:

```
void d2_b1_g(int n, int m,
    double* x, double* d2_x, double* b1_x, double* d2_b1_x,
    double* p, double* d2_p, double* b1_p, double* d2_b1_p,
    double* f, double* d2_f, double* b1_f, double* d2_b1_f)
```

The sub- and superscripts in $v^{(2)}_{(1)}$ are represented by a corresponding concatenation of prefixes d2_b1_v. To compute

$$H_{xx}(x, p, \lambda)x^{(2)} = x^{(2)}_{(1)} =< \lambda, f_{xx}(x, p), x^{(2)} >$$

and

$$H_{xp}(x, p, \lambda)p^{(2)} = p^{(2)}_{(1)} =< \lambda, f_{xp}(x, p), p^{(2)} >$$

we set b1_f=λ and d2_b1_x=d2_b1_p=d2_b1_f=0 in addition to x, d2_x, p, and d2_p. $H_{x\lambda}(x, p)\lambda^{(2)} = f_x(x, p)\lambda^{(2)}$ is computed efficiently using the tangent-linear model of f.

[9] Two columns are structurally orthogonal if they do not have nonzero entries in the same row.

[10] The terms H_{xp} and H_{pp} are derived from the projections $H_{xp}p^{(2)}$ and $H_{pp}p^{(2)}$ where $p^{(2)} = \partial p/\partial p = I$ is the identity matrix.

5. Case studies

Our algorithm is tested using two test problems: one batch reactor model from [19] and a reduced model of a domestic heating system with stratified storage tank which is taken from [20, 21]. The batch reactor example originally stems from the Dow Chemical Company [5]. The reactor is modeled by a differential-algebraic initial value problem of type (2)-(3) with $n = 10$ states, $m = 8$ parameters and a scalar objective function ($k = 1$) . The state vector x represents the time-dependent concentrations, the parameter vector p represents the time-invariant kinetic coefficients.

The domestic heating system is modeled by a set of parametric partial differential-algebraic equations. The partial differential equations are discretized using the method of lines (MOL) [22] resulting in a set of parametric ordinary differential-algebraic equations. The number of state variables strongly depends on the fineness of the MOL discretization. Since we ran the problem with the dense linear algebra options, we used a coarse discretization to totally result in $n = 116$ states and $m = 33$ parameters. We chose an arbitrary scalar objective function ($k = 1$), just to evaluate the speed of the gradient and Hessian computations.

The user provided derivative functions for the forward sweep in eq. (16), and the reverse sweep in eqs. (17) – (19) were implemented using two modes: operator overloading by standard techniques and source code transformation by means of the derivative code compiler `dcc`. In our setting, all functions of the forward sweep are constructed by tangent-linear drivers while all functions of the backward sweep use first- or second-order adjoint drivers.

All computation were performed on notebook running Windows 7 on a 2.53 MHz CPU[11] using the Visual Intel C++ Compiler (version 11.1) using processor specific optimizations. The results for the batch reactor problem are shown in Table 3, the ones for the domestic heating system are presented in Table 4.

For each mode (operator overloading or `dcc`) we measured four values: the time only for the state integration (zeroth-order), for the state and forward sensitivity equations (zeroth- and first-order), for the first-order adjoint backward sweep (first-order) and for the second-order adjoint backward sweep (first- and second-order). The times for the backward sweep do not include the preceding forward sweep, such that e.g. in Table 1 the total time of the batch reactor problem Hessian computation employing `dcc`-generated residual derivatives comprises the 0th- & 1st-order forward and the 1st- & 2nd-order backward sweep to result in a total computational time of 38 ms.

Table 3: Gradient and Hessian computation for batch reactor problem (in milliseconds)

	Forward sweep		Backward sweep	
Mode	only 0th-order	0th- & 1st-order	only 1st-order	1st & 2nd-order
Operator overloading	7 ms	26 ms	15 ms	127 ms
`dcc`	4 ms	13 ms	7 ms	25 ms

Table 4: Gradient and Hessian computation for domestic heating system problem (in seconds)

	Forward sweep		Backward sweep	
Mode	only 0th-order	0th- & 1st-order	only 1st-order	1st & 2nd-order
Operator overloading	0.25 s	2.40 s	0.36 ms	10.2 s
`dcc`	0.26 s	3.50 s	0.36 s	4.30 s

We see that for the first case study, NIXE runs significantly faster with the `dcc`-generated derivatives. In the second case study, overloading outperforms `dcc` for the residual derivatives of the combined zeroth- and first-order forward sweep. For the combined first- and second-order backward sweep, `dcc` is significantly faster.

Moreover, the modified discrete adjoint approach of NIXE seems to be a promising techniques: using `dcc`-generated derivatives for the case studies, the backward sweep costs maximal two times the forward sweep.

[11] Intel(R) Core(TM)2 Duo P9600

6. Summary and outlook

We presented two core computational routines of the AC-SAMMM project: NIXE, a solver for (second-order) adjoint sensitivity analysis of parametric differential-algebraic equations, and `dcc`, the derivative code compiler which provides arbitrary high derivatives of C-code by semantic code transformation. In a case study, `dcc` generated code was interfaced to NIXE and compared with derivative generation by operator overloading. The modified discrete adjoint approach of NIXE seems to be a fast implementation for adjoint sensitivity analysis, since for the two case studies, the backward sweep costs maximal two times the forward sweep.

Future work will focus on algorithmic improvements of NIXE and `dcc`, main extensions regarding the problem scope as well as on the use of Modelica to specify the problem formulation in a high-level language.

References

[1] A. Griewank, A. Walther, Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation, SIAM, Philadelphia, 2008.

[2] C. Bischof, M. Bücker, P. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, Lecture Notes in Computational Science and Engineering, Springer, Berlin, 2008, to appear.

[3] M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (Eds.), Automatic Differentiation: Applications, Theory, and Tools, no. 50 in Lecture Notes in Computational Science and Engineering, Springer, Berlin, 2006.

[4] G. Corliss, C. Faure, A. Griewank, L. Hascoet, U. Naumann (Eds.), Automatic Differentiation of Algorithms – From Simulation to Optimization, Springer, New York, 2002.

[5] G. Blau, L. Kirkby, M. Marks, An industrial kinetics problem for testing nonlinear parameter estimation algorithms, Process Math Modeling Department, The Dow Chemical Company (1981).

[6] G. Dünnebier, D. v. Hessem, J. Kadam, K.-U. Klatt, M. Schlegel, Optimization and control of polymerization processes, Chemical Engineering Technology 28 (5) (2005) 575–580.

[7] P. Betsch, P. Steinmann, A dae approach to flexible multibody dynamics, Multibody System Dynamics 8 (3) (2002) 365–389.
URL `http://dx.doi.org/10.1023/A:1020934000786`

[8] K. E. Brenan, S. L. Campbell, L. R. Petzold, Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations, SIAM, 1996.

[9] Q. Zhang, C. Liu, Dynamical behavior in a differential-algebraic algal blooms model, in: Control Conference, 2008. CCC 2008. 27th Chinese, 2008, pp. 757–761. doi:10.1109/CHICC.2008.4605893.

[10] M. Schlegel, K. Stockmann, T. Binder, W. Marquardt, Dynamic optimization using adaptive control vector parameterization, Computers & Chemical Engineering 29 (8).

[11] P. Deufhard, E. Hairer, J. Zugck, One-step and extrapolation methods for differential- algebraic systems, Numer. Math. 51 (5) (1987) 501–516.
URL `http://dx.doi.org/http://dx.doi.org/10.1007/BF01400352`

[12] E. Hairer, G. Wanner, Solving Ordinary Differential Equations II - Stiff and Differential-Algebraic Problems, Springer, Berlin, 1996.

[13] U. Nowak, Dynamic sparsing in stiff extrapolation methods, IMPACT Comput. Sci. Eng. 5 (1) (1993) 53–74. doi:http://dx.doi.org/10.1006/icse.1993.1003.

[14] M. Schlegel, W. Marquardt, R. Ehrig, U. Nowak, Sensitivity analysis of linearly-implicit differential-algebraic systems by one-step extrapolation, Appl. Numer. Math. 48 (1) (2004) 83–102. doi:http://dx.doi.org/10.1016/j.apnum.2003.07.001.

[15] A. Curtis, M. Powell, J. Reid, On the estimation of sparse Jacobian matrices, Journal of the Institute of Mathematics and Applications 13 (1974) 117–119.

[16] U. Naumann, DAG reversal is NP-complete, Journal of Discrete Algorithms (7) (2009) 402–410.

[17] J. Utke, U. Naumann, C. Wunsch, C. Hill, P. Heimbach, M. Fagan, N. Tallent, M. Strout, OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes, ACM Transactions on Mathematical Software 34 (4).

[18] L. Hascoët, V. Pascual, TAPENADE 2.1 user's guide, Rapport technique 300, INRIA, Sophia Antipolis (2004).
URL `http://www.inria.fr/rrrt/rt-0300.html`

[19] M. Caracotsios, W. Stewart, Sensitivity analysis of initial value problems with mixed odes and algebraic equations, Comp. Chem. Eng. 9 (4) (1985) 359–365.

[20] T. Kreuzinger, Control of a domestic heating system with stratified storage tank, Ph.D. thesis, RWTH Aachen University (2009).

[21] T. Kreuzinger, M. Bitzer, W. Marquardt, Mathematical modelling of a domestic heating system with stratified storage tank, Mathematical and Computer Modelling of Dynamical Systems 14 (3) (2008) 231–248.

[22] W. Schiesser, The numerical method of lines: integration of partial differential equations, Academic Press San Diego, 1991.